AD-A077 525   MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER   F/G 9/4
LOCAL RECONFIGURATION OF NETWORKS OF PROCESSORS: ARRAYS, TREES,--ETC(U)
JUL 79 T DUBITZKI , A WU , A ROSENFELD   AFOSR-77-3271
UNCLASSIFIED   CSC-TR-790   AFOSR-TR-79-1159   NL

1 OF 1
ADA
077525

END
DATE
FILMED
1 -80
DDC

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

14 CSC -TR-790 19 TR-79-1159 11 Jul[redacted]979

18 AFOSR-77-3271

6 LOCAL RECONFIGURATION OF NETWORKS OF PROCESSORS: ARRAYS, TREES, AND GRAPHS.

12 27

10 Tsvi/Dubitzki,
Angela/Wu
Azriel/Rosenfeld

Computer Vision Laboratory
Computer Science Center
University of/Maryland
College Park, MD 20742

9 Interim rept.,

ABSTRACT

This paper studies local reconfiguration of trees into arrays and vice versa. It also studies the construction of adjacency graphs and quadtrees for images stored in cellular array processors.

16 2304 17 A2

15

The support of the U.S. Air Force Office of Scientific Research under Grant AFOSR-77-3271 is gratefully acknowledged, as is the help of Kathryn Riley in preparing this paper.

79 11 23 025

403 018 mt

1. Introduction

In [1] we introduced the concept of local reconfiguration of networks of processors, and presented algorithms for converting between strings and cycles, arrays, trees, etc. Section 2 of this paper discusses direct reconfiguration of trees into arrays and vice versa. Sections 3 and 4 describe how reconfiguration can be used to construct the adjacency graph and quadtree of an image stored in a cellular array.

2. Array/tree Reconfiguration

In the reconfiguration algorithms described in [1], the initial or final graph was always a string. Hence transforming from one configuration to the other, e.g., between a tree and an array, must be done through a string, and thus will take O(N) time, where N is the number of nodes in the graph. In this section we show how to directly convert trees into arrays and arrays into trees in $O(\sqrt{N})$ time.

The problem of using reconfiguration techniques to balance a binary tree in parallel has also been considered. However, it seems to require O(N) time, since O(N) connections may all have to migrate through the root. Thus it is simpler to balance a tree by converting the unbalanced tree into a string and then again into a balanced binary tree by the method described in [1].

Algorithm 2.1: Reconfiguring a two-dimensional array into a minimum-height binary tree.

Let A be a rectangular array of automata (Fig. 1) which contains N nodes where $N = r \cdot s$ ($r \le s$) for integers r,s. D is a distinguished node at the northwest corner of A.

The basic steps of the algorithm are:

(1) Send a signal down from D along the leftmost vertical line. Upon receipt of this signal, each node below D along the vertical line sends a signal to erase the series of horizontal arcs emanating from it in A. This gives us an unbalanced binary tree with height at most $r + s$ (Fig. 2). We can view this tree as composed of one horizontal string of length s and s vertical strings of length $r - 1$. (The distinctions between left, right up and down connections at each node are known in A.)

(2) D sends a signal to order each string to turn into a balanced binary tree in the way described in [1]. This takes at most O(s) time. We now have $r + 1$ binary trees: one with height $O(\lfloor \log s \rfloor)$ and s with height $O(\lfloor \log (r-1) \rfloor)$. In the above process the tree arcs are marked.

(3) Define the tree with s nodes as the "horizontal" tree T and the t trees with (r-1) nodes as "vertical" trees. We will hang the "vertical" trees on the leaves of the horizontal tree T. This is done as follows:

D (Fig. 2) sends a horizontal triggering signal through all the nodes of the tree T in A. Upon arrival at a node i (including D itself) the signal causes node i to check how many marked arcs of the tree are connected to it. If that number is 1 or 2 (except the root of T which is marked and considered as a node with 3 tree arcs) it means that respectively 2 or 1 of the "vertical" trees can be hung on node i in T. Then node i sends (ahead of the triggering signal) a searching signal for 2 or 1 roots of "vertical" trees either through the node below it in A or to the right, checking at each node whether the "vertical" tree below it, in A, is still connected to it. If it is still connected, then it can be assigned to node i of T, i.e. node i connects itself to the roots of its assigned trees and the arcs of A connecting these "vertical" trees to the upper horizontal line of A are disconnected. All the new connecting arcs to the roots of the "vertical" trees are marked as tree arcs. The horizontal triggering signal continues to the right one time unit after the searching signal starts, in order to avoid too many temporary connections at any node of T. In case the above searching signal, starting at node i, does not find enough needed unassigned "vertical" trees to its right, it bounces back to the left in the upper horizontal line of A to look for unassigned "vertical" trees left by the previous searching signals. This is not done when i is the rightmost node in A's top line.

(4) All the unmarked arcs (of A) are erased by a breadth first search signal from D sent down the spanning tree of A.

In the following a leaf is defined to be a node which does not have two sons in T and is said to have one or two null links.

<u>Claim 2.1.1</u>: There are enough null links at the leaves of T to hang all the "vertical" trees in A.

<u>Proof</u>: There are s nodes in T. By induction the number of null links in a binary tree with s nodes is s + 1. On the other hand there are only s "vertical" trees in A.

<u>Corollary</u>: If the rightmost node in A's top line finds under it one unassigned tree to be hung on it, then it doesn't bounce a signal back along A's top line since Claim 2.1.1 proves that there is one less "vertical" tree in A than needed to fill all the null links.

<u>Claim 2.1.2</u>: The height of the combined tree formed from T and the tree hanging from it is at most one unit more than the height of a balanced binary tree formed from a string of $N = s \cdot r$ nodes.

<u>Proof</u>: The height of a balanced binary tree with N nodes is $h = \lfloor \log_2 N \rfloor$. The total height of the combined tree constructed by Algorithm 2.1 will be (see Fig. 3):

$$H = 1 + \lfloor \log_2 s \rfloor + \lfloor \log_2 (r-1) \rfloor \le 1 + \lfloor \log_2 s \rfloor + \lfloor \log_2 r \rfloor \le 1 + \lfloor \log_2 N \rfloor$$

so that $H \le h + 1$.

Claim 2.1.3: Algorithm 2.1 takes O(s) time.

Proof: Step (1) of disconnecting the horizontal lines in A takes O(s+r) time.

Step (2) of converting all the strings into binary trees takes O(s) time.

Step (3) of converting the binary trees into one tree takes O(s) time.

Step (4) of erasing nontree arcs takes O(s+r) time.

Algorithm 2.2: Reconfiguring a complete binary tree into a two-dimensional array.

Let T be a complete binary tree of automata with N nodes. Let D be the root of T. By a complete tree we mean a tree in which all the paths from the root to the leaves are of the same length. In the following a leaf node of T is a node with two null links.

The basic steps of the algorithm are:

(1) Conversion into a tree of strings:

In parallel D sends two signals down T, one at unit speed and the other at 1/3 speed. The unit speed signal bounces back from the leaves of T and meets the 1/3 speed signal at a node in the middle of each path from D to the leaves of T (Fig. 4). Each such meeting node marks itself and turns the subtree rooted at it into a string in the way described in [1]. The unit speed signals continue up to D and make it convert the binary tree rooted at it and having as leaves the marked nodes into a string also. We thus obtain a horizontal string (the last one) with two folded strings hanging from every other node of it (Fig. 5), since in converting a binary tree into a string in the way described in [1], every two leaf nodes are separated by a nonleaf node, and the above twofold strings hang only on leaf nodes. D knows that the process of turning the specified subtrees into strings has terminated as soon as it receives (from its two sons in T) the string generating signals which bounced back from T's leaves. All the arcs of T not participating in the above construction are erased as follows: D send breadth first erasing signals down in T. The signals bounce back from the leaves towards D and on their way back erase every arc of T except the first level of arcs above the leaves and above the marked nodes.

(2) Formation of a pseudo-array:

D orders every hanging point (in the horizontal string of Fig. 5) of a twofold string to order the first node in the right part of the twofold string hanging from it to connect itself to the node to its right and then disconnect itself from its old hanging point. The rightmost node of the horizontal string doesn't have a nonleaf node to its right and therefore orders its right neighbor in the twofold string hanging from it to be a new hanging point to its right (thus part of the horizontal string) from which hangs the rest of the right part of that rightmost twofold string (Fig. 6). We now have a binary tree composed of a set of strings hanging vertically from a horizontal string. This binary tree is a "pseudo-array" and we need only generate the horizontal connections in it in order to get an array. Note that the rightmost hanging string is one node shorter than the other hanging strings (Fig. 6).

(3) Conversion into an array:

First we define for each node in the pseudo-array of step (2) what its upward,downward and horizontal connections are.

For this purpose D sends a breadth-first search signal down the pseudo-array. The signals bounce back from the bottom nodes of the vertical strings (Fig. 6) and go back up in the strings of the pseudo-array. Each entrance to a node in this path is a downward connection and each exit an upward connection. Upon arriving at the marked nodes of step (1) the definitions of the connections change to horizontal until the signals reach D again. Each node in the horizontal line will not emit a signal in the horizontal direction towards D until it has received a horizontal signal. Thus upon receiving two signals D will know that this marking process has terminated. At this stage D orders each of its horizontal neighbors to connect itself temporarily to the node on its downward connection (Fig. 7). Then each of the horizontal neighbors of D orders its vertical neighbors and the node below D to connect themselves. The above temporary connections are then disconnected. In turn each horizontal neighbor of D starts such a connecting process too. This process propagates in the first upper row of the pseudo-array; at the same time each node below that row, having established a horizontal arc, starts such a process in the row below it, and so on until the network of horizontal arcs in D is completed.

<u>Claim 2.2.1</u>: The length of the string formed from the upper part of T (the upper row of the final array) is $O(\sqrt{N})$.

<u>Proof</u>: The number of nodes in T is N which equals $2^{h+1} - 1$ in a complete binary tree with height h. The marked nodes in step (1) of Algorithm 2.2 divide T into an upper complete tree with height h/2 and the rest of T. In that upper part of T we have $N' = 2^{h/2+1} - 1$ nodes. Therefore N' is $O(\sqrt{N})$.

<u>Claim 2.2.2</u>: Each hanging point in the pseudo-array of step (2) of Algorithm 2.2 is the middle of the twofold string hanging from it and the lengths of all the twofold strings in the pseudo-array are equal.

<u>Proof</u>: A complete binary tree has equal numbers of nodes in the right subtree and left subtree of its root. The subtrees forming the twofold strings in step (1) of Algorithm 2.2 are complete binary trees. The process of converting a binary tree into a string [1] produces a string in which the root of the tree is an internal point, all the nodes to its right come from the right subtree of the root and all the nodes to its left come from its left subtree. Thus the root of the tree (a hanging point) is the middle of the twofold string. The lengths of all the twofold strings in the pseudo-array are equal since all the marked nodes of step (1) are at the same depth below D and hence all the subtrees below them are of the same size.

<u>Corollary</u>: The array formed in step (3) of Algorithm 2.2 is of size $O(\sqrt{N}) \times O(\sqrt{N})$. This is due to the fact that the upper horizontal line of the array contains $O(\sqrt{N})$ nodes by Claim 2.2.1

and the lengths of all the vertical strings hanging from the horizontal line of step (2) are equal by Claim 2.2.2.

Note that Algorithm 2.2 is applicable with slight changes to non-complete balanced binary trees. In particular if we are dealing with height-balanced binary trees with minimal numbers of nodes, then the upper horizontal line of the array holds less than $\sqrt{N}$ nodes since the marked nodes of step (1) (closest to the root) are now closer to D than in the case of a complete binary tree because of the existence of short paths going through a node to the leaves of T. Also the difference in length between the vertical hanging strings grows with N since we are dealing with subtrees (generating the twofold strings) which differ more and more in their numbers of nodes as the height of T grows. These factors give us finally very incomplete rectangular arrays. For an example of a height-balanced binary tree with height 6 and a minimal number of nodes see Fig. 8 and 9.

Even the usage of augmented memory at the nodes to balance the twofold strings of Fig. 9, by counting, wouldn't make them equal in length. Trying to balance a tree in parallel involves shifting nodes between its already balanced subtrees and thus involves communication conflicts and rebalancing of the subtrees.

<u>Claim 2.2.3</u>: Algorithm 2.2 takes $O(\sqrt{N})$ time.

<u>Proof</u>: Step (1) of constructing the tree of strings takes $O(\log N)$ time by [1]. Step (2) of constructing the pseudo-array takes constant time. Step (3) of forming the horizontal lines of the array takes $O(\sqrt{N})$ time since we already have a skeleton of an array of size $O(\sqrt{N}) \times O(\sqrt{N})$.

3. Adjacency Graph Construction

Let A be an array (of N processors) in which a picture P is stored, i.e. a value is stored in each node of A. Then A consists of regions S_i each of which is a connected component of constant value. With each S_i we can associate a distinguished node D_i, e.g., the leftmost of its uppermost nodes (note that this node is on S_i's outer border). In this section we show how to reconfigure A so as to construct a graph whose nodes are the D_i's and where two D_i's are joined by an arc iff the corresponding S_i's are adjacent. This graph G is called the adjacency graph of P.

The D_i's can identify themselves by a method described in Lemma 2 of [4], using signals that are sent around the borders of the S_i's. This process is initiated by the distinguished node D of A constructing a spanning tree of A and sending a triggering signal down the tree. The process takes O(perimeter of S_i) time; note that this may be O(N). If we used signals that can travel outside the S_i's, we could find any D_i in O(diameter of S_i) $\leq O(\sqrt{N})$ time; but it is difficult to do this for all the S_i's simultaneously, since many signals may have to pass through the same place. A solution to this problem is contained in a recent paper by Kosaraju [5].

Each D_i now constructs a breadth-first spanning tree T_i of its region S_i. It then sends a signal s around the outer border of S_i which carries along with it a direct link to D_i. At each step, s checks for changes in the values of the adjacent nodes that are not in S_i. Any such change implies that a new region S_j adjacent to S_i has been encountered. (Of course, it may also have been encountered before.) In particular, at the initial step, the non-S_i value (or values) adjacent to D_i defines such an S_j.

Whenever a new S_j is encountered, s splits off a subsidiary signal s' that carries a direct link to D_i up the spanning tree of S_j, while s itself continues around the border of S_i. If necessary, s' waits until the spanning tree of S_j has been completed. While traveling up the tree, s' may also have to wait for other signals that are traveling up it; but the delay is at most O(N). When s' reaches D_j, a direct link has been established between D_i and D_j.

When s has traveled completely around the outer border of S_i, we thus have guaranteed a link between D_i and D_j for every S_j that is adjacent to S_i along its outer border. (If S_j touches S_i several times, several links will be sent from D_i to D_j; but when they arrive, they coalesce, since only one arc is allowed between a given pair of nodes.) The entire process thus takes O(N) time.

For any two adjacent regions S_h and S_k, either S_h is adjacent to the outer border of S_k or vice versa; it is not possible that each of them is inside a hole in the other one, since they cannot surround each other. Hence the construction described in the preceding paragraphs will create a connection between the distinguished nodes of every pair of adjacent regions in O(N) time.

Finally, we indicate how the distinguished node D of A can tell that the process of constructing the adjacency graph is complete. In the process of the D_i's identifying themselves, certain marks exist on the borders of the S_i's. After each D_i identifies itself, it sends s around its border which splits and carries links from D_i to all the neighboring D_j's; thus D_i knows when all of these links have arrived. When this happens, D_i sends an erasing signal down the spanning tree of S_i to eliminate all the marks. Meanwhile, the leaf nodes of the spanning tree of A send back signals toward A's distinguished node D which are stopped by the marks. Thus when the signals reach D, we know that the marks are all gone, which means that the D_i's have all been created and have joined themselves to all their neighboring D_j's.

It should be pointed out that the adjacency graph of P may have nodes of degree O(N), since a large region may have many regions adjacent to it. Thus if the allowable node degree is bounded, we cannot establish direct connections between all pairs of D_i's whose regions are adjacent. Instead, we can pass each signal up the spanning tree until it can go no further, i.e., until the nodes above it have all exhausted their allowed degrees. Thus the distinguished nodes of the regions adjacent to S_j will have direct connections that reach as high as possible in S_j's spanning tree, i.e., that come as close as possible to D_j. Since the number of regions adjacent to S_j cannot exceed its perimeter, which cannot exceed its area, there is room in the tree for all the needed connections.

It is straightforward to compute various properties of each region; examples are area, perimeter, coordinates of centroid, etc. These computations are carried out with the aid of the spanning tree, and take O(tree height) time, which unfortunately may be as high as O(N). The results can be output by the distinguished node of the region, or stored along a path ending at that node (or at the node itself, if augmented memory [3] is allowed). They can then be used as inputs to region merging processes carried out on the graph.

4. <u>Quadtree Construction</u>

The quadtree representation of a picture [6] is obtained by recursively subdividing it into quadrants until blocks of constant value are obtained; this subdivision process is represented by a tree of degree 4. Let A be a rectangular array with N nodes ($N = 2^K \times 2^K$, K integer) in which a picture P is stored, as in Section 3. In this section we show how to reconfigure A to construct the quadtree of P.

First we check whether P has constant value or must be subdivided into quadrants. This is done as follows: The distinguished node D of A (which we assume to be located at its northwest corner) constructs a breadth-first spanning tree of A. The leaves of this tree send their values up the tree. If a node receives different values from its sons, it transmits a "different" signal upward; if it receives the same value from all of them, it transmits that value. After $O(\sqrt{N})$ time, D receives either a "different" signal or a value. In the latter case, the quadtree consists of D itself together with the associated value. In the former case, A must be subdivided.

Subdivision into quadrants is done as follows: D sends a pair of signals to the right along the top row of A, one at unit speed and the other at 1/3 speed. The unit speed signal bounces back from the right end of the row and meets the 1/3 speed signal in the middle of the row. At the same time D sends two signals down the leftmost column of A which meet in the middle of the column. These row and column "midpoints" send unit speed signals downward and rightward, respectively. These signals meet at the "midpoint" of P, which is considered from now on as the new distinguished node $\bar{D}$ of A (Fig. 10). Actually $\bar{D}$ is the southeast corner of the northwest quadrant of A. Also, the row and column through which these signals passed are marked. The subdivision of D into quadrants is by lines below the marked row and to the right of the marked column, respectively (see Fig. 10).

$\bar{D}$ now starts a search for nonuniformity in each of its four quadrants. These searches are analogous to the one made by D, with the obvious modifications to allow for the fact that D is adjacent to, rather than in, three of the quadrants, and is at a different corner of each of them. A spanning tree is constructed in each quadrant, as bounded by the marked row and column, to test for uniformity. At the same time, the distinguished node of each quadrant is found; it is the southeast node of its northwest subquadrant. A direct link is transmitted from $\bar{D}$ to each of these distinguished nodes.

If any quadrant is found to be nonuniform, it is further subdivided as just described. At each step, the "middle" rows and columns are marked so that the signals at subsequent steps can bounce off them and the spanning trees can be confined to them. This process is repeated (in parallel) until no more non-uniformities can be found. This results in a set of uniform-valued blocks, each having a distinguished node that is directly linked to the distinguished node of its parent block (Fig. 10).

It is easy to see that the quadtree construction takes $O(\sqrt{N})$ time. Indeed, at each step, the time required to check a block for uniformity, find its distinguished node, and subdivide it if necessary, is O(block diameter). The successive block diameters are $\sqrt{N}$, $\sqrt{N}/2$, $\sqrt{N}/4$,...; hence the total time is $O(\sqrt{N}) + O(\sqrt{N}/2) + O(\sqrt{N}/4) + ... = O(\sqrt{N})$.

The degree of each node in the quadtree construction process is bounded (see Fig. 10). Indeed, even in the worst case where every node of A is a leaf node, there are only about N/3 nonleaf nodes. Moreover, since the nodes are at "midpoints" of quadrants, it is easily seen that no array node can be a tree node at more than one level above the single pixel level. Hence the nonleaf nodes have degree at most 9 (four brother neighbors in the array; one father neighbor and four son neighbors in the tree), while the leaf nodes have degree at most 5, except that if a leaf node at the single pixel level is also serving as a nonleaf node, it has degree at most 10 (four brothers, two fathers, four sons).

If augmented memory is available, any quadtree leaf node can find the leaf nodes whose blocks are adjacent to its block, e.g. on the east, by proceding as follows: The node sends a signal up the tree, keeping track of its sequence of upward moves, until it arrives at a node from its northwest or southwest son. The signal then travels downward, using the mirror images of the upward moves. If it reaches a leaf by the time the move sequence is exhausted, that leaf is the given leaf's sole eastern neighbor; otherwise, it continues downward and westward, splitting whenever a choice is available, until leaves are reached. The signal can carry with it a direct link to its originating node. Unfortunately, it does not seem to be possible to do this for all leaves at once, since many upward moving signals may collide at the upper levels of the tree.

As a generalization of the quadtree construction process, instead of checking for perfect uniformity in each block, we can compute the mean and standard deviation of its values, and subdivide it only if the standard deviation is high. This can be done in O(block diameter) time if the nodes have augmented memory.

References

[1] A. Wu and A. Rosenfeld, Local Reconfiguration of Networks of Processors. TR-730, Computer Science Center, University of Maryland, February 1979.

[2] D. E. Knuth, The Art of Computer Programming Vol. 3: Sorting and Searching. Addison Wesley, 1975.

[3] T. Dubitzki and A. Wu, Cellular d-graph Automata with Augmented Memory. TR-771, Computer Science Center, University of Maryland, June 1979.

[4] A. R. Smith, Two-Dimensional Formal Languages and Pattern Recognition by Cellular Automata, 12th Annual Symposium on Switching and Automata Theory, 1971, 144-152.

[5] S. R. Kosaraju, Fast parallel processing array algorithms for some graph problems, Proc. 11th ACM Symp. on Theory of Computing, 1979, 231-236.

[6] C. R. Dyer, A. Rosenfeld and H. Samet, Region representation: boundary codes from quadtrees. TR-732, Computer Science Center, University of Maryland, February 1979.

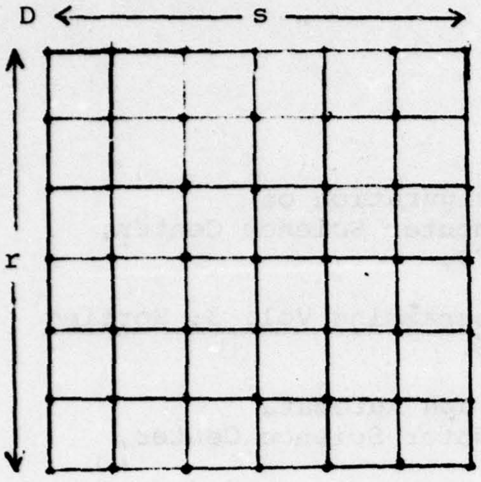
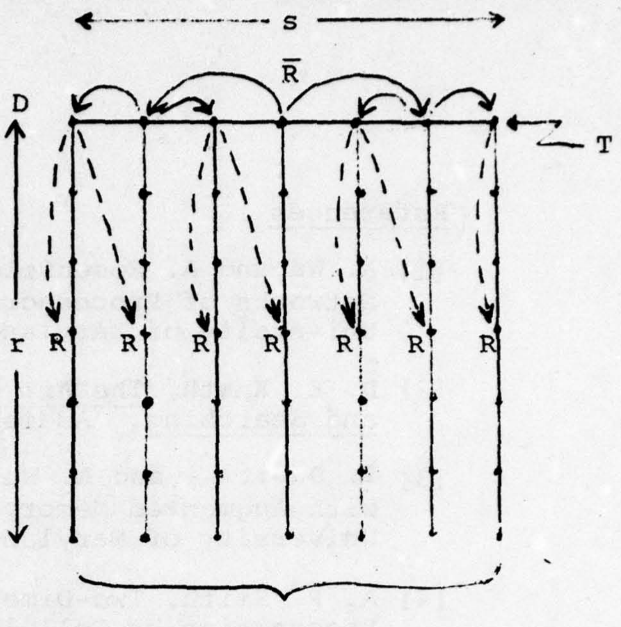


Fig. 1



"vertical" trees

Fig. 2

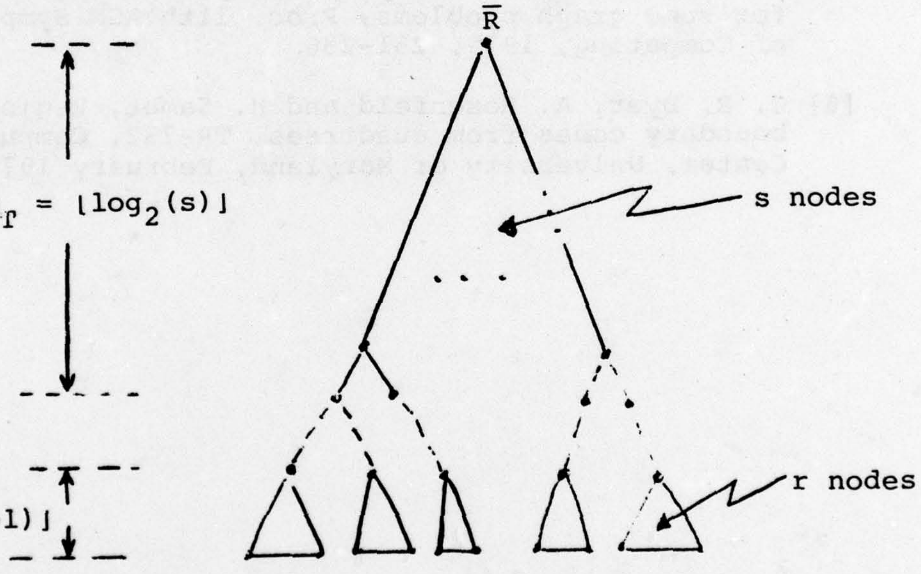

$\bar{R}$
$h_T = \lfloor \log_2(s) \rfloor$
s nodes
$h = \lfloor \log_2(r-1) \rfloor$
r nodes

Fig. 3

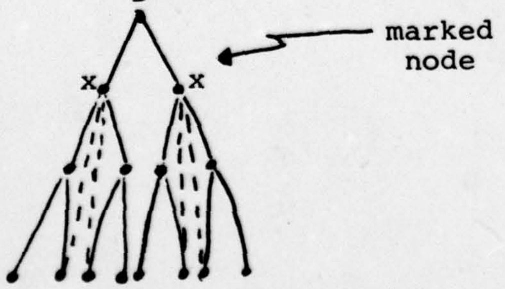


Fig. 4

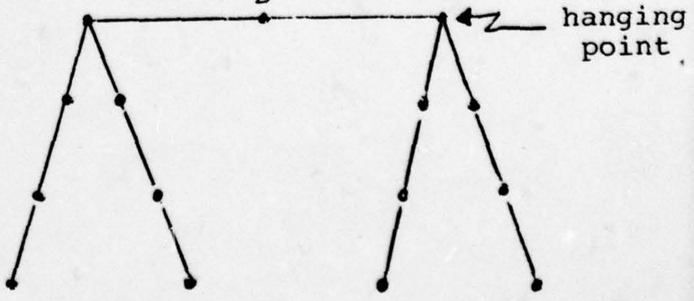


Fig. 5

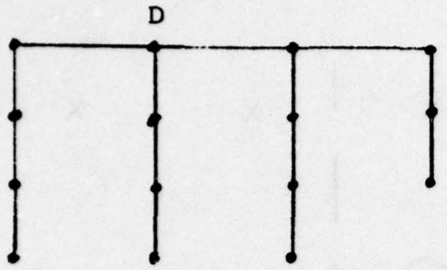


Fig. 6

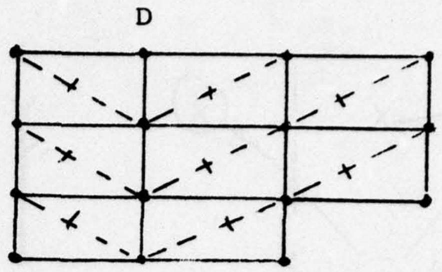


Fig. 7

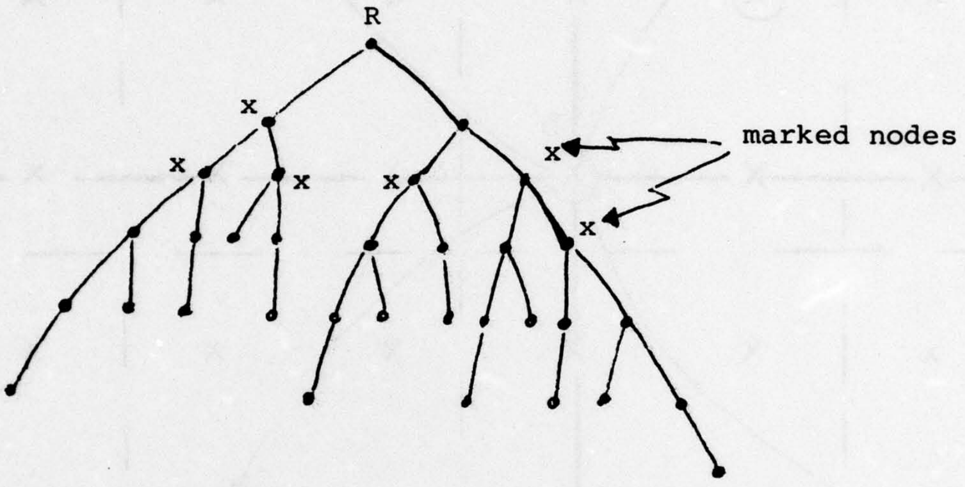


Fig. 8

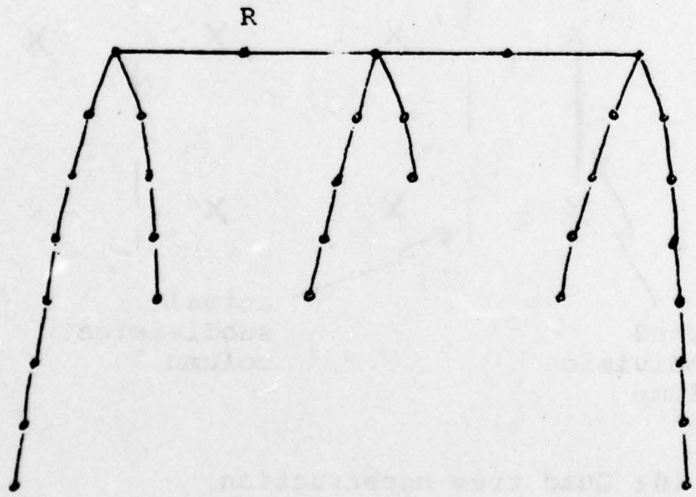


Fig. 9

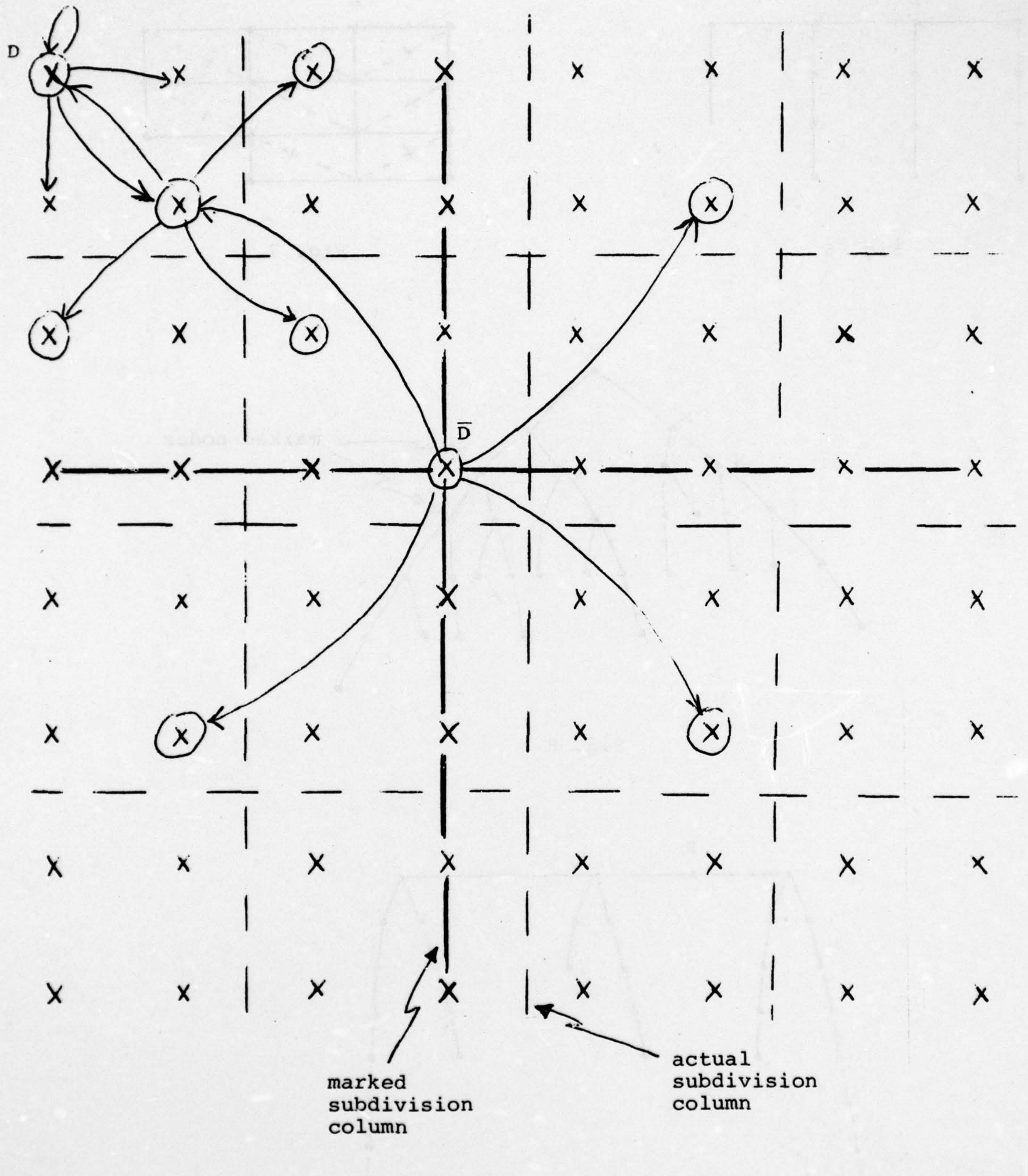


Fig. 10 : Quad tree construction.
ⓧ --roots of subtrees in the quad tree.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) Unclassified

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 79-1159	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LOCAL RECONFIGURATION OF NETWORKS OF PROCESSORS: ARRAYS, TREES, AND GRAPHS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER TR-790
7. AUTHOR(s) Tsvi Dubitzki, Angela Wu, Azriel Rosenfeld		8. CONTRACT OR GRANT NUMBER(s) AFOSR-77-3271
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Vision Laboratory, Computer Science Center, University of Maryland, College Park, MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM, Bolling AFB, Washington, DC 20332		12. REPORT DATE July 1979
		13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Networks
Cellular automata
Trees
Arrays

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper studies local reconfiguration of trees into arrays and vice versa. It also studies the construction of adjacency graphs and quadtrees for images stored in cellular array processors.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)